Q1: What sort of design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

_

Answer: I will use Strategy design pattern. There will be an abstract base class **Hydra**, which contains some common gaming operations (e.g. play()). The Hydra game that is specified in the project description will be implemented as a child class **BasicHydra** which is inherited from **Hydra**. In main function, we will create the game by a pointer to the class **Hydra**, and assign it to the actual game type we want it to be. When we change the interface(s), we will only need to change them in class declarations and definitions, but not in main function. When we want to change the rules, we can either make changes to **BasicHydra** or create a new child class inherited from **Hydra**. In the latter case, the only change made in main function is that the pointer will be allocated to the new game class we created.

Q2: Jokers have a different behaviour from any other card. How should you structure your card type to support this without special-casing jokers everywhere they are used?

Answer: I will use a class **Card** to represent all cards except jokers and a class **Joker** which inherit from class **Card** to represent jokers. Class **Joker** will have its own public method which sets its card value to the specific value. This method can only be called once since as long as the value is chosen for a joker, it cannot be changed afterward. This way, I can use a (smart) pointer to **Card** to represent all cards and simply call **Joker**'s public method when needed without special-casing jokers due to C++'s dynamic dispatch.

Q3: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

Answer: To allow computer players, I will use a class **ComputerPlayer** which is inherited from the concrete class **Player**, which represents actual players. I will use Strategy design pattern to handle dynamic strategies of a computer player. i.e. There will be some classes containing decision making strategies, and they all have the same parent class. **ComputerPlayer** will have a pointer to that parent class so that it can be dynamically changed. To make the changes on play strategies of computer players, I will use implement a public method which does the change, and this will affect further decisions the computer player makes.

Q4: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

Answer: First, I will use PImpl design pattern to make all private fields contained in a pointer to PImpl. Then I will add a public method for class **Player**, which takes a pointer (probably a shared pointer) to **ComputerPlayer** as a parameter. It transfers its pointer PImpl (i.e. draw pile, discard pile, currentCard, reserve, etc.) to the **ComputerPlayer**. The advantage of using PImpl is that we don't have to change the signature of the "accepting" method of **ComputerPlayer** when some fields of **Player** change in the future.

Breakdown of the project

I will complete the whole project design class by class and from bottom to top. My aim is to complete define and implement all functions of the specified classes. It's also expected that I may change the interfaces of the previous class(es) when implementing further classes (i.e. When implementing **BasicHydra**, I realize I need to change the definition of some function in **Head**). However, this is omitted in the following plan, and the estimated completion date has already taken care of it.

Aug 5: Class **Card, Joker, Pile**

Aug 6: Class **Head, Heads, Player**

Aug 7-8: Class **Hydra, BasicHydra**

Aug 9: main function; debug with some testing cases.

Aug 10: Debugging continued. If things go well, can start designing class **ComputerPlayer** and think about how to design **Strategy** class and subclasses of **Strategy**.

Aug 11-12: Debugging with some testing on **ComputerPlayer**. If things go well, can start thinking about what other enhancements are possible.

Aug 13-15: Debugging with new enhancements. Finalize documentations. Make sure no fatal errors exist in the code. Make sure the code will compile and run.